Universidade do Minho
Escola de Engenharia

Projeto de Arquiteturas de Software
4º ano de MIEI

Refactoring

Relatório de Desenvolvimento

José André Martins Pereira
(a82880@alunos.uminho.pt)

Ricardo André Gomes Petronilho
(a81744@alunos.uminho.pt)

16 de Janeiro de 2020

Conteúdo

Capítulo 1

Introdução

Na unidade curricular de Arquiteturas de Software, foi-nos proposta a identificação de *code smells* e *refactoring* dos mesmos, num código fornecido pela equipa docente.

Na verdade, o código fornecido pela equipa docente, foi desenvolvido por colegas do curso, para o primeiro projeto da unidade curricular, consistindo numa plataforma de *Trading*.

Ao longo deste relatório, aborda-se os tipos de *code smells* encontrados, e os tratamentos aplicados aos mesmos, e as respetivas técnicas de *refactoring* utilizadas. Ainda são determinadas e calculadas algumas métricas que inflênciam a estrutura do código.

O objetivo principal deste projeto, consiste na perceção dos problemas que existem com o desenvolvimento de código, sendo estes responsáveis por dificuldades encontradas nos processos de análise, debugging e adição de novas funcionalidades à aplicação.

Por fim, este projeto também se torna importante, para se refletir dos *code smells* também cometidos no primeiro projeto desenvolvido pelo grupo, nesta unidade curricular. Sendo assim, o conhecimento destes *code smells* é necessário, para não serem novamente cometidos.

Capítulo 2

Identificação e Refactoring de Code Smells

Neste capítulo identificam-se os diferentes *code smells* presentes no código fornecido pela equipa docente e os respetivos tratamentos efetuados (refactoring).

2.1 Bloaters

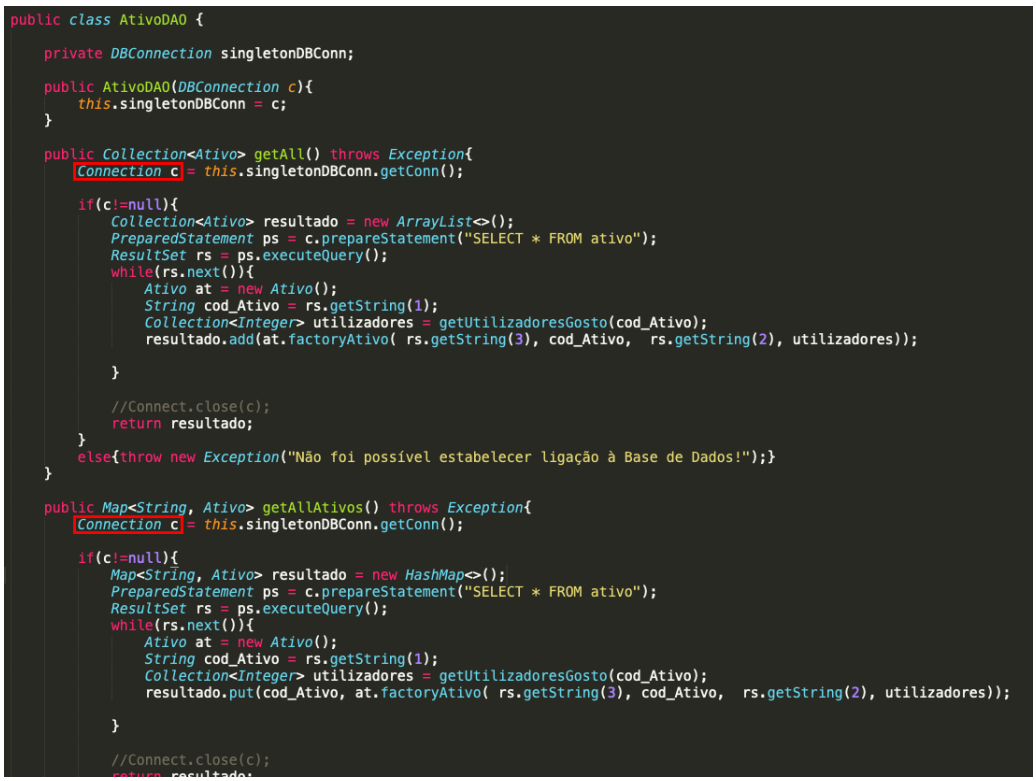
Nesta secção apresentam-se os diferentes tipos de *Bloaters* encontrados. Um *code smell Bloater* consiste em código, métodos e classes de grandes proporções, dificultando a sua análise.

2.1.1 Data Clumps

Os *Data Clumps* ocorrem quando grupos de variáveis ocorrem repetitivamente ao longo do código. Desta forma, no código fornecido, nas classes *DAO* verificou-se a repetição de uma variável **c** do tipo *Connection* em todas os métodos, tal como se pode observar na figura 2.1.

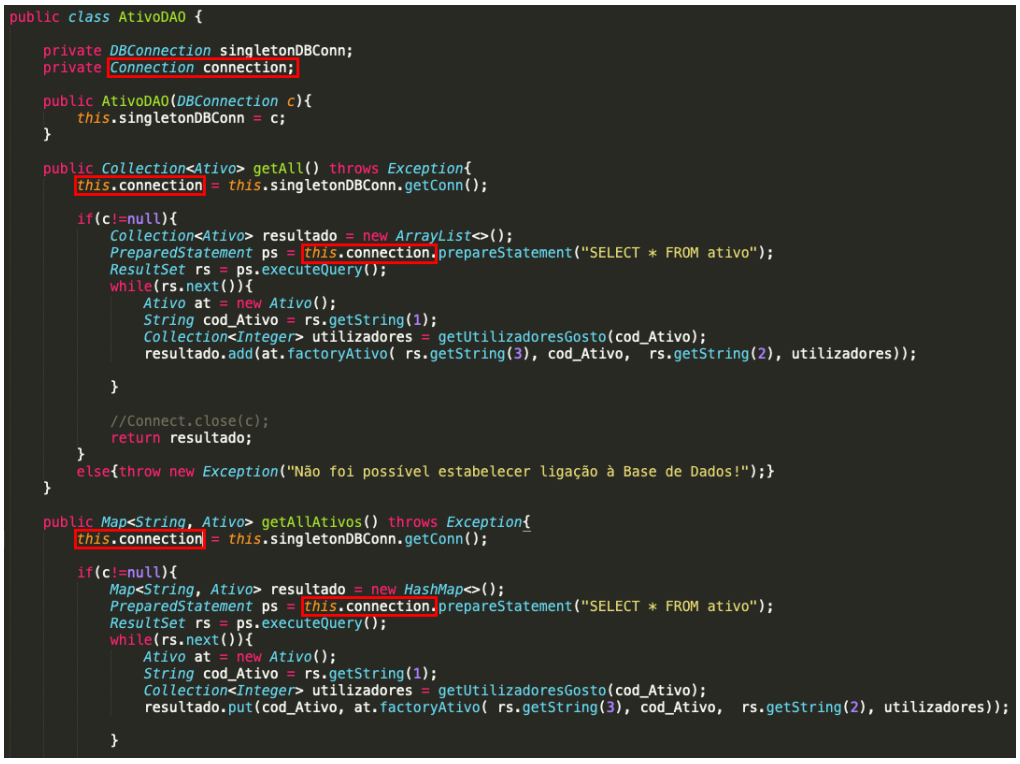
Assim, o tratamento realizado a este *code smell*, foi a declaração desta variável **c**, como variável de instância da classe **AtivoDAO** e a colocação de um nome mais intuitivo para a variável, como **connection**, tal como se pode observar na figura 2.2.

O *code smell*, ocorre em todas as classes *DAO*, ou seja, o tratamento vai ser aplicado a todas.



```
public class AtivoDAO {

    private DBConnection singletonDBConn;

    public AtivoDAO(DBConnection c){
        this.singletonDBConn = c;
    }

    public Collection<Ativo> getAll() throws Exception{
        Connection c = this.singletonDBConn.getConn();

        if(c!=null){
            Collection<Ativo> resultado = new ArrayList<>();
            PreparedStatement ps = c.prepareStatement("SELECT * FROM ativo");
            ResultSet rs = ps.executeQuery();
            while(rs.next()){
                Ativo at = new Ativo();
                String cod_Ativo = rs.getString(1);
                Collection<Integer> utilizadores = getUtilizadoresGosto(cod_Ativo);
                resultado.add(at.factoryAtivo( rs.getString(3), cod_Ativo,  rs.getString(2), utilizadores));

            }

            //Connect.close(c);
            return resultado;
        }
        else{throw new Exception("Não foi possível estabelecer ligação à Base de Dados!");}
    }

    public Map<String, Ativo> getAllAtivos() throws Exception{
        Connection c = this.singletonDBConn.getConn();

        if(c!=null){
            Map<String, Ativo> resultado = new HashMap<>();
            PreparedStatement ps = c.prepareStatement("SELECT * FROM ativo");
            ResultSet rs = ps.executeQuery();
            while(rs.next()){
                Ativo at = new Ativo();
                String cod_Ativo = rs.getString(1);
                Collection<Integer> utilizadores = getUtilizadoresGosto(cod_Ativo);
                resultado.put(cod_Ativo, at.factoryAtivo( rs.getString(3), cod_Ativo,  rs.getString(2), utilizadores));

            }

            //Connect.close(c);
            return resultado;
```

Figura 2.1: Repetição da variável **c** to tipo *Connection* em todos os métodos.



```
public class AtivoDAO {

    private DBConnection singletonDBConn;
    private Connection connection;

    public AtivoDAO(DBConnection c){
        this.singletonDBConn = c;
    }

    public Collection<Ativo> getAll() throws Exception{
        this.connection = this.singletonDBConn.getConn();

        if(c!=null){
            Collection<Ativo> resultado = new ArrayList<>();
            PreparedStatement ps = this.connection.prepareStatement("SELECT * FROM ativo");
            ResultSet rs = ps.executeQuery();
            while(rs.next()){
                Ativo at = new Ativo();
                String cod_Ativo = rs.getString(1);
                Collection<Integer> utilizadores = getUtilizadoresGosto(cod_Ativo);
                resultado.add(at.factoryAtivo( rs.getString(3), cod_Ativo,  rs.getString(2), utilizadores));

            }

            //Connect.close(c);
            return resultado;
        }
        else{throw new Exception("Não foi possível estabelecer ligação à Base de Dados!");}
    }

    public Map<String, Ativo> getAllAtivos() throws Exception{
        this.connection = this.singletonDBConn.getConn();

        if(c!=null){
            Map<String, Ativo> resultado = new HashMap<>();
            PreparedStatement ps = this.connection.prepareStatement("SELECT * FROM ativo");
            ResultSet rs = ps.executeQuery();
            while(rs.next()){
                Ativo at = new Ativo();
                String cod_Ativo = rs.getString(1);
                Collection<Integer> utilizadores = getUtilizadoresGosto(cod_Ativo);
                resultado.put(cod_Ativo, at.factoryAtivo( rs.getString(3), cod_Ativo,  rs.getString(2), utilizadores));

            }
```

Figura 2.2: Tratamento realizado ao *code smell* data clumbs da categoria Bloaters.

2.1.2 Long Method

```
    @Override
    public void run() {
        while (true){

            try {
                sleep(5000);
                fd.atualizaAtivos();
                Collection<Ativo> ativos = fd.getAtivosToDC();
                //Collection<ContratoAberto> results = this.contratos.getContratosAbertosFromUtilizador(this.idUtilizador);


                if (!ativos.isEmpty()) {

                    ativos.stream().forEach(h -> {

                        try {
                            String codAtivo = h.getCodigo_API();
                            Collection<ContratoAberto> results = this.fd.getContratosFromAtivo(codAtivo);

                            double precoNovo = 0;
                            if(!results.isEmpty() || h.hasUtilizadores()) precoNovo = fd.getPrecoAtivoDouble(codAtivo);
                            double finalPrecoNovo = precoNovo;

                            if(h.hasUtilizadores()){
                                Collection<Integer> utilizadores = h.getUtilizadores();
                                utilizadores.stream().forEach(ut -> {
                                    try {
                                        double precoAntigo = fd.getValorFromGosto(ut, codAtivo);
                                        double mudanca = Math.abs(finalPrecoNovo-precoAntigo)/precoAntigo;
                                        if(mudanca >= mudanca_significativa){
                                            System.out.println("vou notificar thread");
                                            h.notifyObservers(ut, precoAntigo, finalPrecoNovo);
                                            System.out.println("vou notificar thread");
                                        }

                                    } catch (Exception e) {
                                        e.printStackTrace();
                                    }

                                });
                            }
```

```
                            results.stream().forEach(y -> {
                                try {


                                    double stopLoss = y.getSL();
                                    double takeProfit = y.getTP();
                                    int key = y.getId();

                                    double balanco = y.getBalanco(finalPrecoNovo);

                                    if (balanco >= takeProfit || balanco <= stopLoss)
                                        this.fd.addContratoHistorico(y.getIdUtilizador(), key);

                                }catch(Exception e){
                                    e.printStackTrace();
                                }

                            });
                        }catch(Exception e){
                            e.printStackTrace();
                        }
                    });
                }

            }catch (Exception e){
                e.printStackTrace();
            }
        }
    }
}
```

Figura 2.3: Um método com 67 linhas.

O *code smell* identificado na figura 2.3, consiste num método que contém mais de dez linhas de código, dificultando a sua leitura e dessa forma a sua análise.

Assim, o tratamento para este *code smell*, denomina-se *Extract Method*, consiste em distribuir diferentes partes/funcionalidades do método, em métodos auxiliares mais simples, e com responsabilidades distintas, reduzindo assim, o seu tamanho e facilitando a sua análise. Deste modo, foram criados três métodos auxiliares e removidas/renomeadas certas variáveis que não fariam sentido.

No bloco de código, que ficou no método **verificarPrecoAtivosSeguidos**, verificou-se que estava a ser usada um variável desnecessária, denominada **finalPriceNovo**, deste modo, a mesma foi removida.

```
@Override
    public void run() {
        while (true){

            try {
                sleep(5000);
                fd.atualizaAtivos();
                Collection<Ativo> ativos = fd.getAtivosToDC();

                if (!ativos.isEmpty()) verificaAtivosSeguidos(ativos);

            }catch (Exception e){
                e.printStackTrace();
            }
        }
    }
```

Figura 2.4: Método run simplificado.

```
public void verificaAtivosSeguidos(Collection<Ativo> ativos){
    ativos.stream().forEach(ativo -> {
        try {
            String codAtivo = ativo.getCodigo_API();
            Collection<ContratoAberto> contratos = this.fd.getContratosFromAtivo(codAtivo);

            verificarPrecoAtivosSeguidos(codAtivo, contratos, ativo);

            contratos.stream().forEach(contratoAberto -> {verificaStopLossTakeProfit(contratoAberto);});
        }catch(Exception e){
            e.printStackTrace();
        }
    });
}
```

Figura 2.5: Método auxiliar, que por sua vez chama os outros métodos auxiliares.

```
public void verificarPrecoAtivosSeguidos(String codAtivo, Collection<ContratoAberto> contratos, Ativo ativo){
    double priceNovo = 0;
    if(!contratos.isEmpty() || ativo.hasUtilizadores()) priceNovo = fd.getPrecoAtivoDouble(codAtivo);

    if(ativo.hasUtilizadores()){
        Collection<Integer> utilizadores = ativo.getUtilizadores();
        utilizadores.stream().forEach(ut -> {
            try {
                double priceAntigo = fd.getValorFromGosto(ut, codAtivo);
                double mudanca = Math.abs(priceNovo-priceAntigo)/priceAntigo;
                if(mudanca >= mudanca_significativa){
                    ativo.notifyObservers(ut, priceAntigo, priceNovo);
                }

            } catch (Exception e) {
                e.printStackTrace();
            }

        });
    }
}

public void verificaStopLossTakeProfit(ContratoAberto contratoAberto) throws Exception{
    try {
        double stopLoss = contratoAberto.getSL();
        double takeProfit = contratoAberto.getTP();
        int key = contratoAberto.getId();
        double balanco = contratoAberto.getBalanco(finalPriceNovo);

        if (balanco >= takeProfit || balanco <= stopLoss)
            this.fd.addContratoHistorico(contratoAberto.getIdUtilizador(), key);
        }catch(Exception e){
            e.printStackTrace();
        }
}
```

Figura 2.6: Métodos auxiliares chamados no método **verificarAtivosSeguidos**.

2.1.3 Long Parameter List

O *code smell* identificado consiste em métodos com mais de quatro argumentos. Desta forma, encontrou-se um método, denominado **addContratoPortfolio**, na class **Facade**, do package **Negócio**, que tem mais de quatro argumentos, tal como se pode observar na figura 2.7.

```
public void addContratoPortfolio(int idU, String codAtivo, double montante, double quantidade,
                                 int leverage, boolean compra, double TP, double SL, double valorAtivo) throws Exception {

    if ((montanteMaximo(idU) - montante - SL) > 0) {
        Utilizador user = utilizadores.get(idU);
        double newSaldo = user.getCapital() - montante;
        utilizadores.updateCapital(idU, newSaldo);
        ContratoAberto inserir;
        if(compra)  inserir = new ContratoAbertoCompra(0, montante, LocalDateTime.now(), quantidade, codAtivo, leverage,  TP,  SL, valorAtivo, i
        else inserir = new ContratoAbertoVenda(0, montante, LocalDateTime.now(), quantidade, codAtivo, leverage,  TP,  SL, valorAtivo, idU);
        contratosAbertos.put(inserir);
    }else {throw new Exception("Não tem saldo suficiente na conta.");}
}
```

Figura 2.7: Método com mais de quatro argumentos.

Deste modo, como se trata de uma adição do contrato ao portfolio, o tratamento a efetuar denomina-se *Introduce Parameter Object*, ou seja, como o objeto **Contrato** contém quase todas as variáveis enviadas como argumento, excepto o *StopLess* SL, *Take Profit* TP e o tipo de contrato, *Compra* ou *Venda*, então envia-se o objeto *Contrato*, e as variáveis que faltam neste, tal como se pode observar na figura 2.8.

Neste tratamento também foi aplicado a técnica *Extrat Variable*, visto que as variáveis **quantidade**, **montante**, entre outras, são utilizadas mais do que uma vez, compensando guardar o valor das mesmas numa variável temporária, para que não se chame repetitivamente o método *get* dessas variáveis.

Importante realçar que, ainda se podia ter feito um melhor tratamento, mas devido à hierarquia do **Contrato**, não estar correta, pois tal como se irá abordar mais adiante, não era necessário a diferenciação entre contratos abertos ou fechados, ou de compra e venda através de classes, pois estes vários tipos ou derivações de contratos, mantém as mesmas variáveis.

Na verdade, o correto na opinião do grupo, será diferenciar estes tipos, com a utilização de variáveis na classe **Contrato** e dessa forma, este método apenas precisaria de receber o objeto **Contrato**. Isto é, bastava colocar na classe **Contrato** um variável boolean, que identifica se o contrato é de compra ou venda, e outra variável boolean, que identifica se o contrato está aberto ou fechado.

```
public void addContratoPortfolio(Contrato contrato, double TP, double SL, boolean tipo) throws Exception {
    int idU = contrato.getIdUtilizador();
    double montante = contrato.getMontantePago();
    double quantidade = contrato.getQuantidade();
    String codAtivo = contrato.getCodAtivo();
    int leverage = contrato.getLeverage();
    double valorAtivo = contrato.getValorAbertura();

    if ((montanteMaximo(idU) - montante - SL) > 0) {
        Utilizador user = utilizadores.get(idU);
        double newSaldo = user.getCapital() - montante;
        utilizadores.updateCapital(idU, newSaldo);
        ContratoAberto inserir;
        if(tipo)  inserir = new ContratoAbertoCompra(0, montante, LocalDateTime.now(), quantidade, codAtivo, leverage,  TP,  SL, valorAtivo, idU
        else inserir = new ContratoAbertoVenda(0, montante, LocalDateTime.now(), quantidade, codAtivo, leverage,  TP,  SL, valorAtivo, idU);
        contratosAbertos.put(inserir);
    }else {throw new Exception("Não tem saldo suficiente na conta.");}
}
```

Figura 2.8: Método reduzido para 4 argumentos.

2.2 Object-Orientation Abusers

Nesta secção apresentam-se os diferentes tipos de *code smells* desta categoria. Um *Object-Orientation Abusers* consiste na aplicação incorreta ou incompleta dos princípios de programação orientada a objetos.

2.2.1 Design pattern observer incorretamente implementado

Verificou-se que a equipa não implementou o *design pattern observer* da melhor forma.

Este *design pattern* consiste num objeto (Observable) notificar outro (Observer), através do método **notifyObservers()**. A equipa que desenvolveu o código implementou essa lógica, no entanto **especificou todos os parâmetros** que são enviados na notificação para o Observer. Desta forma, caso sejam necessários enviar mais parâmetros, facilmente o método notifyObservers() fica com uma **lista desnecessariamente aumentada de parâmetros**, tornando-se num *code smell* denominado de *Long Parameter List*, tal como a seguinte figura ilustra.

```
public void notifyObservers(int idUt, double valorAntigo, double valorNovo) throws Exception {
    for(Observer o : observadores){
        String msg = "O ativo " + designacao + " sofreu uma alteração significativa ! Passou de " + valorAntigo + " para " + valorNovo;
        o.update(this.id, idUt,  msg, valorNovo)
    }
}
```

Figura 2.9: Incorreta especificação dos parâmetros referentes ao Ativo no método notifyObservers.

Acrescentando, ao especificar os parâmetros (neste caso sobre o Ativo) torna a reutilização da interface impossível (que vai contra um dos princípios de POO) uma vez que o código não é genérico. Desta forma propõe-se a alteração da lista de parâmetros específicos ao Ativo, para apenas um único argumento do tipo Object, tornando o *design pattern* genérico e possível de reutilizar as interfaces várias vezes ao longo do projeto ou mesmo noutros projetos, tal como se apresenta de seguida.

```
public void notifyObservers(Object arg) {
    for(Observer o : observadores){
        o.update(arg);
    }
}
```

Figura 2.10: Modificação dos parâmetros para um único argumento genérico do tipo Object.

Desta forma a evocação do método notifyObservers() também sofreu alterações para introduzir a forma mais genérica, antes o método era evocado da forma, como se pode ver na figura 2.11.

```
ativo.notifyObservers(ut, priceAntigo, finalPriceNovo);
```

Figura 2.11: Evocação do método notifyObservers antes da modificação.

Para corrigir a modificação, cria-se um array de objetos contendo os parâmetros e envia-se os mesmos como um único objeto, tal como se pode observar na figura 2.12.

```
Object[] args = new Object[]{ut, priceAntigo, finalPriceNovo};
ativo.notifyObservers(args);
```

Figura 2.12: Evocação do método notifyObservers após a modificação.

2.2.2 Classe com a responsabilidade errada

Na classe Ativo verificou-se que a variável **utilizadores** não faz sentido existir, uma vez que não é da responsabilidade da classe Ativo guardar a lista de utilizadores que seguem um determinado ativo, pois o comportamento de seguir um ativo é realizado pelo utilizador.

Note-se que o código é funcionalmente correto, no entanto um dos princípios de POO é colocar o comportamento (métodos) e variáveis na classe que tem a responsabilidade em causa. Por isso, entendeu-se que a variável utilizadores deve ser removida da classe Ativo e na classe **Utilizador** deve ser introduzida a variável **ativos**, que contém a lista de ativos que o utilizador segue, tal como as figuras 2.13 e 2.14 ilustram.

```
public class Ativo implements Subject {

    private TaxaBehavior taxabehavior;
    private String id;
    private String designacao;

    private Collection<Integer> utilizadores;

    private List<Observer> observadores;
```

Figura 2.13: Variável utilizadores removida da classe Ativo.

```
public class Utilizador {

    private int id;
    private String username;
    private String email;
    private String password;
    private String NIF;
    private double capital;

    private Collection<String> ativos;
```

Figura 2.14: Variável ativos introduzida na classe Utilizador.

2.3 Change Preventers & Couplers

O *Change Preventers* ocorre quando a necessidade de uma alteração numa parte do código, cria a necessidade de se alterar também em muitas outras partes.

O *Couplers* ocorre quando existe acoplamento excessivo entre classes, ou em alternativa a este, excessiva delegação.

A razão pela qual se juntou estes dois *code smells*, deve-se ao facto, de ambos estarem relacionados, isto é, quando um ocorre, o outro provavelmente também. Desta forma, os exemplos encontrados verificam ambos, devido ao facto, por exemplo, acoplamento em excesso, dificultar alterações no futuro.

2.3.1 Shotgun Surgery

Em relação a *code smells* destas categorias, a equipa que desenvolveu o código não seguiu corretamente a noção de package, divisão de responsabilidades e redução das dependências.

Na verdade, o package **DataObject**, não tem uma classe **Facade**, isto é, uma classe que expõe as funcionalidades deste package, encapsulando o interior do mesmo. A vantagem deste encapsulamento, é a redução das dependências entre os packages, permitindo alterações no futuro, sem a necessidade de mudar outros packages ou classes que usem estas funcionalidades.

Na figura 2.15, verifica-se a presença destes *code smells*, isto é, a dependência de classes internas do package **DataObject**, na classe **Facade** do package **Negócio**, que poderia ser facilmente evitada, com a utilização de uma classe **FacadeDataObject** no package **DataObject**.

Assim, se no futuro, fosse necessário mudar totalmente a forma como o package **DataObject** funciona, isto é, as suas classes (por exemplo o nome das mesmas, ou serem substituídas por outras), a classe **Facade** do package **Negócio**, entre outras, também teriam que ser mudadas.

Desta forma, o tratamento a realizar, e como já foi referido, consiste na criação de uma classe no package **DataObject**, que exponha as funcionalidades do package, encapsulando o interior do mesmo. De seguida, adiciona-se essa classe, como variável de instância da classe **Facade** do package **Negócio**, tal como se pode observar na figura 2.16.

```java
public class Facade implements Observer {

    private UtilizadorDAO utilizadores;
    private AtivoDAO ativos;
    private ContratoFechadoDAO contratosFechados;
    private ContratoAbertoDAO contratosAbertos;
    private SaldoCorretoraDAO saldoCorretora;
    private FinanceAPI financeAPI;
    private Map<String, Ativo> mapAtivos;
    private static double plafond = 20;
    private static double taxaLeverage = 0.005;
    private static DecimalFormat df2 = new DecimalFormat("#.##");

    public Facade() throws Exception {
        DBConnection singletonDBConn = DBConnection.getInstance();
        this.saldoCorretora = new SaldoCorretoraDAO(singletonDBConn);
        this.financeAPI = new YahooFinanceAPI();
        utilizadores = new UtilizadorDAO(singletonDBConn);
        ativos = new AtivoDAO(singletonDBConn);
        contratosFechados = new ContratoFechadoDAO(singletonDBConn);
        contratosAbertos = new ContratoAbertoDAO(singletonDBConn);
        atualizaAtivos();
        Thread dealChecker = new Thread(new DealChecker(this));
        dealChecker.start();
    }
```

Figura 2.15: Classe com várias dependências com o package *DataObject*

```java
public class Facade implements Observer {

    private FacadeDataObject data;
    private FinanceAPI financeAPI;
    private Map<String, Ativo> mapAtivos;
    private static double plafond = 20;
    private static double taxaLeverage = 0.005;
    private static DecimalFormat df2 = new DecimalFormat("#.##");
```

Figura 2.16: Classe com apenas uma dependência em relação ao package **DataObject**.

2.3.2 Design pattern factory incorretamente implementado

O *design pattern* Factory Method é utilizado para fabricar objetos de um determinado tipo através de uma interface mais geral, sendo possível produzir diferentes sub-tipos dos mesmos objetos através das respetivas sub-classes. A grande vantagem deste *design pattern* é a **eliminação de dependências com as várias sub-classes**. No caso da classe Ativo caso se pretendesse introduzir mais N sub-classes da mesma, mantinha-se apenas as duas dependências (com a interface IAtivo e a classe FactoryAtivo) ao invés de o número de dependências aumentar em N.

Posto isto, verificou-se que o propósito do *design pattern* não foi de todo respeitado com a implementação atual do código, uma vez que o método de criação do Ativo encontra-se exatamente na mesma classe (Ativo), sendo incorreto, pois este devia estar definido numa classe à parte denominada de FactoryAtivo e é necessário a existência de uma interface IAtivo. Desta forma, o propósito de reduzir o número de dependências não é impossível de atingir, não existindo, na prática, a implementação do *design pattern*, consequentemente aumentado o acoplamento entre as várias classes que usam a classe Ativo.

Conclui-se que o *refactoring*, neste caso, seria mais dispendioso do que simplesmente deixar a versão

atual implementado, uma vez que a própria arquitetura em geral teria de ser alterada, introduzidas interfaces para cada tipo de dados (Ativo, Utilizador e cada tipo de contratos) e não só implementar o design pattern para a classe Ativo como para as restantes.

2.4 Dispensables

Dispensables consiste no código ou documentação que não faz sentido existir ou a ausência do mesmo torna mais eficiente e facilitada a compreensão do código.

Na classe Utilizador verificou-se que a variável NIF não é utilizada tal como a figura 2.17 ilustra.

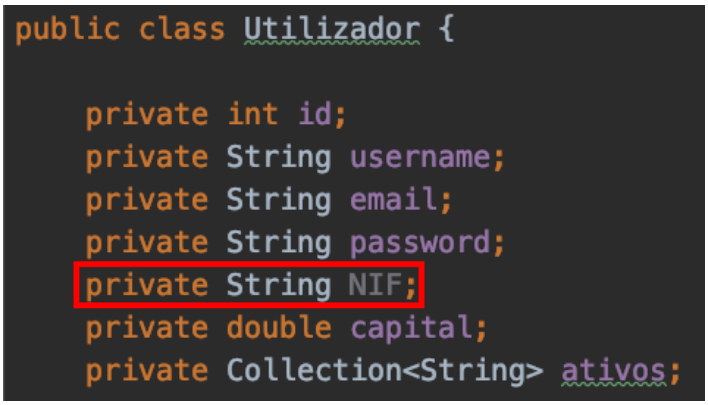


Figura 2.17: Variável NIF não é utilizada na classe Utilizador.

Ainda na mesma classe verificou-se que 6 métodos, 1 getter e 5 setters, não estavam a ser utilizados tal como se pode observar na figura 2.18.

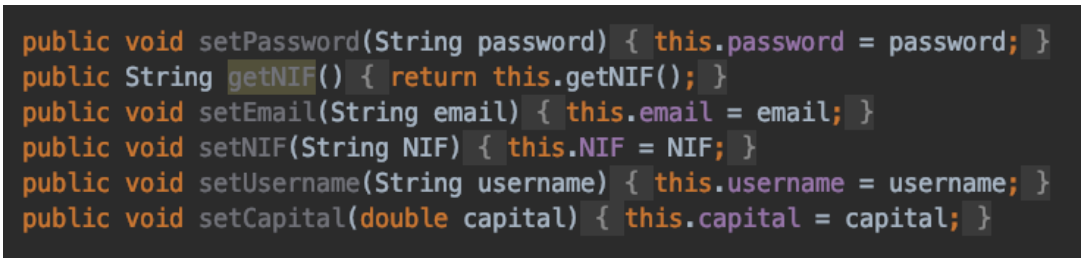


Figura 2.18: 6 métodos não são utilizados na classe Utilizador.

Note-se que, apesar dos métodos não serem utilizados, as variáveis (à excepção do NIF) são utilizadas, por exemplo no construtor. Acrescentando, apesar de no final do trabalho os métodos acabaram por não ser utilizados, tem sentido a sua criação inicial uma vez que é um bom princípio de POO, no momento de criação de uma variável também serem criados os métodos getters e setters, pois podem ser utilizados no futuro.

2.5 Outros

Nesta secção apresentam-se *code smells*, que não tem uma categoria específica, mas que são considerados.

2.5.1 Nome da variável não intuitivo

No desenvolvimento do código, por vezes não se escolhe o nome mais intuitivo para descrever o propósito de uma variável. Apesar do código ser funcionalmente correto, torna mais difícil a compreensão futura do mesmo, e até a própria pessoa que o desenvolveu pode ter dificuldades.

Na figura 2.19, a variável **codigo_API** refere-se ao identificador da classe Ativo, sendo que o nome mais apropriado poderia ser **id**. Note-se que ao se chamar **codigo_API** implicitamente está a indicar que utilizou uma API para obter o Ativo, e por isso não existe abstração de implementação (um dos princípios de POO) e até mesmo porque a classe Ativo pode ser reutilizada para outro projeto (outro príncipio de POO) e deve ser o mais genérica possível sendo que não deve ser alusiva ao uso de uma API. Por outro lado, um **código** pode ser usado para variados propósitos, sendo que deve ser especificado que é o código **identificador**.

```
public class Ativo implements Subject {

    private TaxaBehavior taxabehavior;
    private String codigo_API;
    private String designacao;
    private Collection<Integer> utilizadores;
    private List<Observer> observadores;
```

Figura 2.19: Variável codigo_API na classe Ativo.

```
public class Ativo implements Subject {

    private TaxaBehavior taxabehavior;
    private String id;
    private String designacao;
    private Collection<Integer> utilizadores;
    private List<Observer> observadores;
```

Figura 2.20: Modificação do nome para id.

Na classe DealChecker, responsável por encerrar contratos CFD, reparou-se que cada ativo era denominado por **h** o que, em primeira análise, não se entendeu ao que esta variável se referia, tal como se pode observar na figura 2.21.

```
ativos.stream().forEach(h -> {

    try {
        String codAtivo = h.getCodigo_API();
        Collection<ContratoAberto> results = this.fd.getContratosFromAtivo(codAtivo);

        double priceNovo = 0;
        if(!results.isEmpty() || h.hasUtilizadores()) priceNovo = fd.getPrecoAtivoDouble(codAtivo);
        double finalPriceNovo = priceNovo;

        if(h.hasUtilizadores()){
            Collection<Integer> utilizadores = h.getUtilizadores();
            utilizadores.stream().forEach(ut -> {
                try {
                    double priceAntigo = fd.getValorFromGosto(ut, codAtivo);
                    double mudanca = Math.abs(finalPriceNovo-priceAntigo)/priceAntigo;
                    if(mudanca >= mudanca_significativa){
                        System.out.println("vou notificar thread");
                        h.notifyObservers(ut, priceAntigo, finalPriceNovo);
                        System.out.println("vou notificar thread");
                    }
```

Figura 2.21: Variável h na classe DealChecker.

A simples mudança do nome da variável tornou o código mais fácil de se entender.

```
ativos.stream().forEach(ativo -> {

    try {
        String codAtivo = ativo.getCodigo_API();
        Collection<ContratoAberto> results = this.fd.getContratosFromAtivo(codAtivo);

        double priceNovo = 0;
        if(!results.isEmpty() || ativo.hasUtilizadores()) priceNovo = fd.getPrecoAtivoDouble(codAtivo);
        double finalPriceNovo = priceNovo;

        if(ativo.hasUtilizadores()){
            Collection<Integer> utilizadores = ativo.getUtilizadores();
            utilizadores.stream().forEach(ut -> {
                try {
                    double priceAntigo = fd.getValorFromGosto(ut, codAtivo);
                    double mudanca = Math.abs(finalPriceNovo-priceAntigo)/priceAntigo;
                    if(mudanca >= mudanca_significativa){
                        System.out.println("vou notificar thread");
                        ativo.notifyObservers(ut, priceAntigo, finalPriceNovo);
                        System.out.println("vou notificar thread");
                    }
```

Figura 2.22: Modificação do nome para ativo.

Capítulo 3

Métricas

Neste capítulo aborda-se o conjunto de métricas de código que possam ter impacto nas alterações estruturais do código.

3.1 Média do número de argumentos

A métrica proposta pelo grupo, contabiliza a média de argumentos dos métodos alterados pelo *refactoring*. Desta forma, os métodos alterados foram:

- notifyObservers() da classe **Ativo** no package **Models**;
- update() da classe **Facade** no package **Negócio**;
- addContratoPortfolio() da classe **Facade** no package **Negócio**;

método alterado	# args antes	# args depois
notifyObservers	4	1
update	4	1
addContratoPortfolio	9	4
total de argumentos	17	6
média de argumentos	5.67	2

Tabela 3.1: Número de argumentos antes e depois do *refactoring*.

Observando a amostra conclui-se que houve uma **redução de 64.73 %** do número de argumentos após o *refactoring*.

3.2 Média do número de linhas por método

A métrica proposta pelo grupo, contabiliza a média do número de linhas por método alterado pelo *refactoring*. No entanto, apenas um método sofreu essa redução, o **run** da classe **DealChecker**, no package **Negócio**.

Inicialmente o método **run** tinha sessenta e sete linhas, após *refactoring*, passou para onze linhas. Assim, concluí-se que houve uma redução de 83.58 %.

3.3 Número de classes & interfaces alteradas

A métrica proposta pelo grupo, contabiliza o número de classes alteradas pelas técnicas de *refactoring*.

Desta forma, foram alteradas todas as classes **DAO**, devido a um *code smell* identificado no capítulo anterior, sendo cinco classes.

Do mesmo modo, no package **Models**, foram alteradas duas interfaces, **Observer** e **Subject** e duas classes, **Ativo** e **Utilizador**.

No package **Negócio**, foram alteradas as duas classes, **DealChecker** e **Facade**.

Assim, contabiliza-se a modificação de onze classes/interfaces, podendo ter sido mais, caso fosse feito *refactoring* a todo o código.

Note-se que, esta métrica torna-se importante, para concluir, que a arquitetura foi bastante alterada.

3.4 Número de design patterns alterados

A métrica proposta pelo grupo, contabiliza o número de *desing patterns* alterados ou mesmo completamente modificados devido à incorreta implementação dos mesmos.

Desta forma, tal como referido no capítulo anterior, modificou-se os *design patterns* Observer e Factory para a classe **Ativo**.

Tal como a métrica anterior, esta métrica tem bastante relevância, uma vez que, a alteração de *design patterns* exige sempre muitas alterações na estrutura do código. No caso do *desing pattern* **Factory**, tal como referido no capítulo anterior, o *refactoring* torna-se mais dispendioso, do que implementar de raiz o próprio *desing pattern*.

Capítulo 4

Conclusão

Em suma, conclui-se que os objetivos inicialmente propostos foram atingidos. Desta forma, conseguiu-se identificar todos os tipos de *code smells* e aplicar as técnicas de *refactoring* adequadas.

A análise inicial do código fornecido, foi complicada, pois a arquitetura do mesmo, não estava correta, não sendo perceptível algumas das decisões tomadas pelos desenvolvedores do mesmo, quanto à utilização de hierarquia. Do mesmo modo, também percebeu-se que alguns dos *design patterns*, não estavam bem aplicados.

Uma das principais conclusões a que se chegou, foi que, devido à elevada identificação de *code smells*, provavelmente o *refactoring* seria mais dispendioso a nível de tempo, do que, fazer a aplicação de novo, pois existem muitas dependências no código fornecido, tornando difícil a sua alteração.

Finalizado este trabalho, percebe-se a importância da identificação destes *code smells*, sendo crucial, após o seu conhecimento, tentar evitar que eles ocorram nos nossos códigos, para prevenir os problemas que os mesmos causam.